Einsoft 'APPROVED'

HISOFT PASCAL

FOR USE WITH Einstein COLOUR MICRO COMPUTER

KUMA

HISOFT PASCAL

FOR

THE

TATUNG EINSTEIN MICRO-COMPUTER

ISBN NO 07457-0124-8

TATUNG CODE NO 17-0025-1

Published by:- Kuma Computers Ltd.,
12 Horseshoe Park,
Pangbourne,
Berks RG8 7JW

Telex 849462 Tel 07357 4335

CONTENTS

First Published 1984.

SECTION 0 PRELIMINARIES.

0.0 Introduction.

Hisoft Pascal is a fast, easy-to-use and powerful version of the Pascal language as specified in the Pascal User Manual and Report (Jensen/Wirth Second Edition). Omissions from this specification are as follows:

Only FILEs of CHAR are allowed.
A RECORD type may not have a VARIANT part.
PROCEDUREs and FUNCTIONs are not valid as parameters.

Some extra functions and procedures are included to reflect the changing environment in which compilers are used; among these are POKE, PEEK, CPM and ADDR. Also included are routines to access the graphics and sound facilities of the Tatung Einstein.

The compiler occupies approximately 12K of storage while the runtimes take up roughly 4K. Both are supplied on microfloppy disk in one package.

0.1 Scope of this manual.

This manual is not intended to teach you Pascal; you are referred to the excellent books given in the Bibliography if you are a newcomer to programming in Pascal.

This manual is a reference document, detailing the particular features of Hisoft Pascal. Section 1 gives the syntax and the semantics expected by the compiler.

Section 2 details the various predefined identifiers that are available within Hisoft Pascal, from CONSTants to FUNCTIONs.

Section 3 contains information on the various compiler options available and also on the format of comments.

The above Sections should be read carefully by all users.

Appendix 1 details the error messages generated both by the compiler and the runtimes.

Appendix 2 lists the predefined identifiers and reserved words.

Appendix 3 gives details on the internal representation of data within Hisoft Pascal - useful for programmers who wish to get their hands dirty.

Appendix 4 gives some example Pascal programs - study this if you experience any problems in writing Hisoft Pascal programs.

0.2 Copying Hisoft Pascal onto a work disc.

To prepare a disc for use with Hisoft Pascal, you should first format a disc using the DOS BACKUP command. Then copy onto this disc COPY.COM using the COPY command. See the 'DOS/MOS Introduction' manual for details. For example, on a one drive system you would use:

```
COPY 0:COPY.COM TO 0:
```

If you have a text editor or wordprocessor you should also copy this onto the work disc. Otherwise copy XBAS onto the disc as you will be using this to type in your Pascal programs.

Finally you should copy Hisoft Pascal from the distribution disc using for example:

```
COPY 0:HPEIN.COM TO 0:
```

Now you can put the distribution disc away in a safe place.

0.3 Compiling and Running.

When using Hisoft Pascal on the Tatung Einstein you can type in your Pascal programs using a text editor or wordprocessor. If however you do not possess such a program you can use XBAS to enter source and save it using the SAVE command with an .ASC extension.

For example type the following:

```
XBAS

10 PROGRAM FACT;
20 VAR I:INTEGER;
30 FUNCTION FACT(N:INTEGER):REAL;
40 BEGIN
50 IF N>1 THEN FACT:=N*FACT(N-1) ELSE FACT:=1
60 END;
70 BEGIN
80 REPEAT
90 WRITE('FACTORIAL: ');
100 READLN; READ(I);
110 WRITELN(' IS ',FACT(I));
120 WRITELN
130 UNTIL FALSE
140 END.

SAVE "FACT.ASC"

DOS
```

and then to the 0: prompt

```
HPEIN FACT
```

The source code will be compiled and assuming no errors are detected an object file named FACT.COM will be created. This can be run by simply typing FACT to the DOS prompt. If there are any errors you should hit 'E' to exit the compiler. Then load XBAS and the program and correct the typing mistake.

When this program is run it will prompt you to enter numbers and give the factorial of each number input until you type CTRL/C which will return you to DOS.

Note that on the Tatung Einstien '←' and '→' are used instead of '[' and ']' because the computer does not use a proper ASCII character set. The equivalents of '{' and '}' should not be used for comments as BASIC treats them as tokens. The '(*' and '*)' form must always be used instead. See Section 3.

In the example above we used the simplest form of command line.

The full form looks like this:

```
HPEIN <file1> file2 <;option<,option,....>>
```

where:

file2 is the source file which must have extension of ASC. If this file does not exist then an error message is given.

file1 is an optionally different obect file (with extension COM).

option is an ordinary compiler option (see Sectin 3.2 or one of the following:

N	- don't produce any object code.
Y	- delete any existing object code.
T	- include trig routines and EXP,LN and FRAC in the runtimes.
R	- do not include REAL routines in the run-times.
Vnnnn	- set the runtime stack to nnnn (hexadecimal).
B	- treat the source file a a non-BASIC file. i.e. without line numbers.

The B option is used only when a text editor or wordprocessor is used to create source text. This should be standard CP/M ASCII text. That is lines are terminated by CR,LF and the file is terminated with a CTRL/Z. If you have a wordprocessor that normally uses the top bit of characters you should make sure that this feature is disabled. If the B option is not used the first 7 characters of each line are taken to be the line number.

The T and R options can used to keep the runtimes fairly small if REALs or trigonometric functions are not used. If the latter are used without including them in the runtimes then an '*ERROR* 3' is given. Note use the R option with care since the compiler makes no checks.

Options within the source program override those specified in the command line. The N, Y, T, R, V and B options are not available within the program.

Some example command lines are given below:

HPEIN HILBERT;P+,N

get a compilation listing (to printer) of a program.

HPEIN BYTE;R,Y,D-,C-,S-,A-

turn off all checks; say for a benchmark.

HPEIN OBJ1 SCE1;R,B,L-

compile the program SCE1.ASC entered using a wordprocessor to produce the file OBJ1.COM with no compiler listing (to make compilation faster) and not including REAL routines with the object file.

The compiler generates a listing of the form:

xxxx nnnn text of source line

where: xxxx is the address where the code generated by this line begins.
nnnn is the line number with leading zeroes suppressed.

If a line contains more than 80 characters then the compiler inserts new-line characters so that the length of a line is never more than 80 characters.

The listing may be directed to a printer, if required, by the use of option P+ (see Section 3).

You may pause the listing at any stage by holding the CONTROL key down and the 'S' key down at the same time; use CONTROL and 'C' to return to DOS or any other key to restart the listing.

If an error is detected during the compilation then the message '*ERROR*' will be displayed, followed by an up-arrow ('^'), which points after the symbol which generated the error, and an error number (see Appendix 1). The listing will pause; hit 'E' to return to DOS tidily (deleting the object file), CONTROL and 'C' to abort, or any other key to continue the compilation.

If the program terminates incorrectly (e.g. without 'END.') then the message 'No more text' will be displayed and control returned to DOS.

If a compilation contains any errors then the number of errors detected will be given and any object code produced will be deleted. Otherwise a COM file will be generated (unless you use option N) ready for direct execution from DOS. The COM file contains the object code produced by the compilation and, automatically, the runtime support routines. Remember that you can cut down the size of these runtime routines by using options in the command line - see above.

During a run of the object code various runtime error messages may be generated (see Appendix 1). You may suspend a run while it is outputting to the screen or printer by using CONTROL and 'S'; subsequently use CONTROL and 'C' to abort the run and return to DOS or any other key to resume the run.

0.4 Strong TYPEing.

Different languages have different ways of ensuring that the user does not use an element of data in a manner which is inconsistent with its definition.

At one end of the scale there is machine code where no checks whatever are made on the TYPE of variable being referenced. Next we have a language like the Byte 'Tiny Pascal' in which character, integer and Boolean data may be freely mixed without generating errors. Further up the scale comes BASIC which distinguishes between integers and strings and, sometimes, between integers and reals (perhaps using the '%' sign to denote integers). Then comes Pascal which goes as far as allowing distinct user-enumerated types. At the top of the scale (at present) is a language like ADA in which one can define different, incompatible numeric types.

There are basically two approaches used by Pascal implementations to strength of typing; structural equivalence or name equivalence. Hisoft Pascal uses name equivalence for RECORDs and ARRAYs. The consequences of this are clarified in Section 1 - let it suffice to give an example here; say two variables are defined as follows:

```
VAR A : ARRAY['A'..'C'] OF INTEGER;
    B : ARRAY['A'..'C'] OF INTEGER;
```

then one might be tempted to think that one could write A:=B; but this would generate an error (*ERROR* 10) under Hisoft Pascal since two separate 'TYPE records' have been created by the above definitions. In other words, the user has not taken the decision that A and B should represent the same type of data. She/He could do this by:

```
VAR A,B : ARRAY['A'..'C'] OF INTEGER;
```

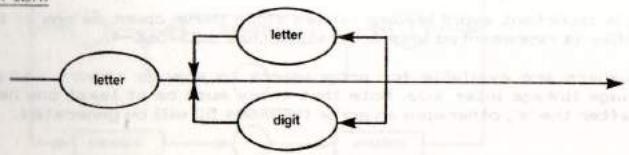
and now the user can freely assign A to B and vice versa since only one 'TYPE record' has been created.

Although on the surface this name equivalence approach may seem a little complicated, in general it leads to fewer programming errors since it requires more initial thought from the programmer.

SECTION 1 SYNTAX AND SEMANTICS.

This section details the syntax and the semantics of Hisoft Pascal - unless otherwise stated the implementation is as specified in the Pascal User Manual and Report Second Edition (Jensen/Wirth).

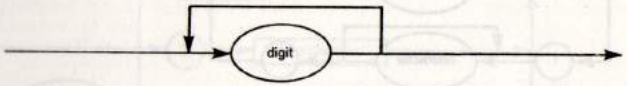
1.1 IDENTIFIER.



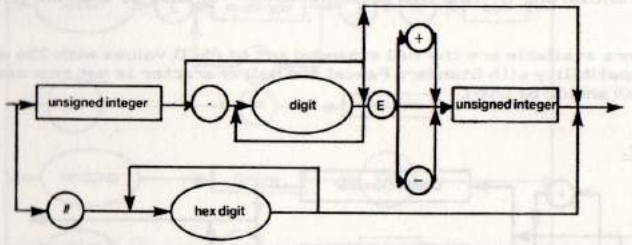
Only the first 8 characters of an identifier are treated as significant. These first 8 characters must not constitute a reserved word (see Appendix 2).

Identifiers may contain lower or upper case letters. Lower case is not converted to upper case so that the identifiers HELLO, HELlo and hello are all different. Reserved words and predefined identifiers may only be entered in upper case.

1.2 UNSIGNED INTEGER.



1.3 UNSIGNED NUMBER.



Integers have an absolute value less than or equal to 32767 in Hisoft Pascal. Larger whole numbers are treated as reals.

The mantissa of reals is 23 bits in length. The accuracy attained using reals is

therefore about 7 significant figures. Note that accuracy is lost if the result of a calculation is much less than the absolute values of its arguments e.g. 2.00002 - 2 does not yield 0.00002. This is due to the inaccuracy involved in representing decimal fractions as binary fractions. It does not occur when integers of moderate size are represented as reals e.g. 200002 - 200000 = 2 exactly.

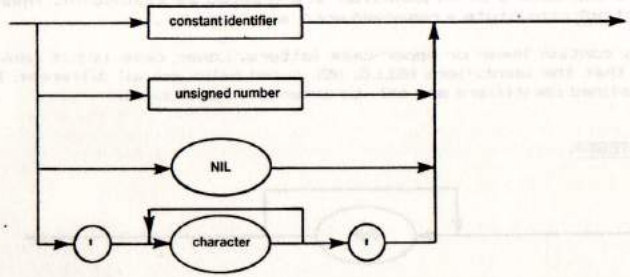
The largest real available is 3.4E38 while the smallest is 5.9E-39.

There is no point in using more than 7 digits in the mantissa when specifying reals since extra digits are ignored except for their place value.

When accuracy is important avoid leading zeroes since these count as one of the digits. Thus 0.000123456 is represented less accurately than 1.23456E-4.

Hexadecimal numbers are available for programmers to specify memory addresses for assembly language linkage inter alia. Note that there must be at least one hexadecimal digit present after the '#', otherwise an error (*ERROR* 51) will be generated.

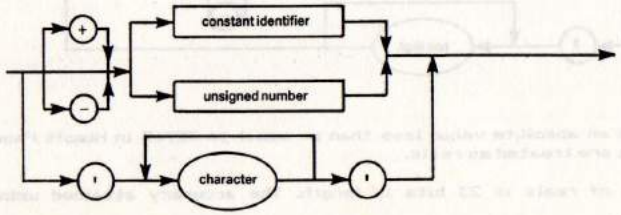
1.4 UNSIGNED CONSTANT.



Note that strings may not contain more than 255 characters. String types are ARRAY [1..N] OF CHAR where N is an integer between 1 and 255 inclusive. Literal strings should not contain end-of-line characters (CHR(13)) - if they do then an '*ERROR* 68' is generated.

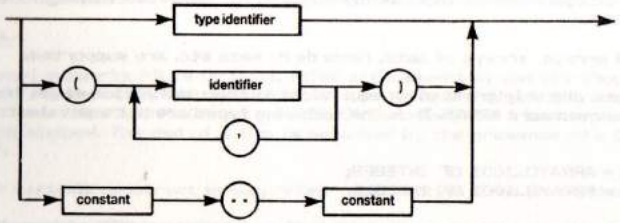
The characters available are the full expanded set of ASCII values with 256 elements. To maintain compatibility with Standard Pascal the null character is not represented as ''; instead CHR(0) should be used.

1.5 CONSTANT.



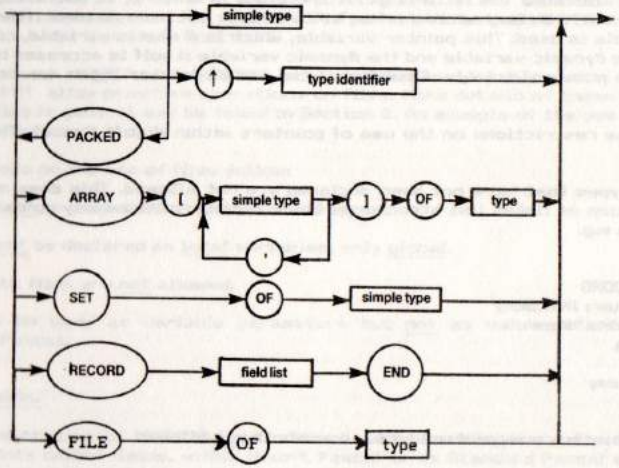
The comments made in Section 1.4 concerning strings apply here.

1.6 SIMPLE TYPE.



Scalar enumerated types (identifier, identifier, ...) may not have more than 256 elements.

1.7 TYPE.



The reserved word PACKED is accepted but ignored since packing already takes place for arrays of characters etc. The only case in which the packing of arrays would be advantageous is with an array of Booleans - but this is more naturally expressed as a

set when packing is required.

1.7.1 ARRAYs and SETs.

The base type of a set may have up to 256 elements. This enables SETs of CHAR to be declared together with SETs of any user enumerated type. Note, however, that only subranges of integers can be used as base types. All subsets of integers are treated as sets of 0..255.

Full arrays of arrays, arrays of sets, records of sets etc. are supported.

Two ARRAY types are only treated as equivalent if their definition stems from the same use of the reserved word ARRAY. Thus the following types are not equivalent:

```
TYPE
      tablea = ARRAY[1..100] OF INTEGER;
      tableb = ARRAY[1..100] OF INTEGER;
```

So a variable of type tablea may not be assigned to a variable of type tableb. This enables mistakes to be detected such as assigning two tables representing different data. The above restriction does not hold for the special case of arrays of a string type, since arrays of this type are always used to represent similar data.

1.7.2 Pointers.

Hisoft Pascal allows the creation of dynamic variables through the use of the Standard Procedure NEW (see Section 2). A dynamic variable, unlike a static variable which has memory space allocated for it throughout the block in which it is declared, cannot be referenced directly through an identifier since it does not have an identifier; instead a pointer variable is used. This pointer variable, which is a static variable, contains the address of the dynamic variable and the dynamic variable itself is accessed by including a '^' after the pointer variable. Examples of the use of pointer types can be studied in Appendix 4.

There are some restrictions on the use of pointers within Hisoft Pascal. These are as follows:

Pointers to types that have not been declared are not allowed. This does not prevent the construction of linked list structures since type definitions may contain pointers to themselves e.g.

```
TYPE
    item = RECORD
           value : INTEGER;
           next  : ^item
          END;

    link = ^item;
```

Pointers to pointers and pointers to files are both <u>not</u> allowed.

Pointers to the same type are regarded as equivalent e.g.

```
VAR
    first : link;
```



```
    current : ^item;
```

The variables first and current are equivalent (i.e. structural equivalence is used) and may be assigned to each other or compared.

The predefined constant NIL is supported and when this is assigned to a pointer variable then the pointer variable is deemed to contain no address.

1.7.3 FILEs.

Hisoft Pascal supports FILEs OF CHAR. Files are sequential and are thought to be made up of lines, separated by a line separator; the lines are physically separated by the codes CRLF (i.e. CHR(13), CHR(10)) but only the CR is treated as a line separator, the LF is ignored and skipped. The end of a file is detected by the presence of a CTRL/Z (CHR(26)) character.

The buffer variable construct is supported – see Section 2 for more details.

The Standard Type TEXT is defined as FILE OF CHAR so that the declarations

```
file1 : FILE OF CHAR;
file2 : TEXT;
```

are equivalent.

Two files are predefined viz. INPUT and OUTPUT. These are textfiles which represent the standard I/O media of the computer i.e. the keyboard (via DOS routine 10) and the CRT (via DOS routine 2). They are the default values in any textfile operations e.g. WRITE(value); is equivalent to WRITE(OUTPUT,value);. Note that INPUT is considered to begin with a blank line – see Section 2.

The procedures RESET and REWRITE are provided to open files while the procedures GET and PUT allow primitive operations on files. More details of these procedures and file handling in general may be found in Section 2. An example of the use of files may be found in Appendix 4.

Restrictions on the use of files follow:

Files may <u>not</u> be components of structured types.

Files <u>cannot</u> be declared as <u>local</u> variables, only <u>global</u>.

Pointers to files are <u>not</u> allowed.

Files may be used as variable parameters but <u>not</u> as value parameters – this is Standard Pascal.

1.7.4 RECORDs.

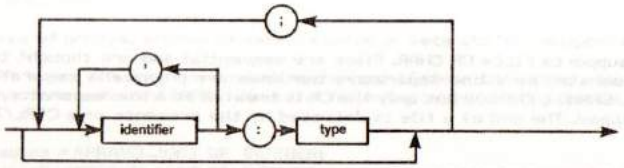
The implementation of RECORDs, structured variables composed of a fixed number of constituents called fields, within Hisoft Pascal is as Standard Pascal except that the variant part of the field list is <u>not</u> supported.

Two RECORD types are only treated as equivalent if their declaration stems from the same occurrence of the reserved word RECORD see Section 1.7.1 above.

The WITH statement may be used to access the different fields within a record in a more compact form.

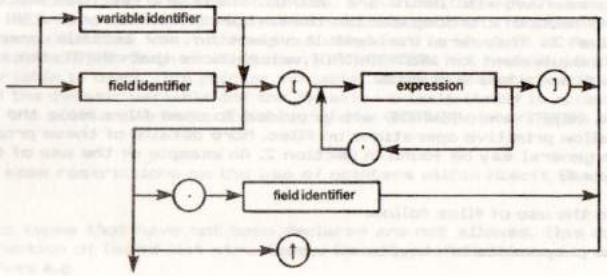
See Appendix 4 for an example of the use of WITH and RECORDs in general.

1.8 FIELD LIST.



Used in conjunction with RECORDs see Section 1.7.4 above and Appendix 4 for an example.

1.9 VARIABLE.



Two kinds of variables are supported within Hisoft Pascal; static and dynamic variables. Static variables are explicitly declared through VAR and memory is allocated for them during the entire execution of the block in which they were declared.

Dynamic variables, however, are created dynamically during program execution by the procedure NEW. They are not declared explicitly and cannot be referenced by an identifier. They are referenced indirectly by a static variable of type pointer, which contains the address of the dynamic variable.

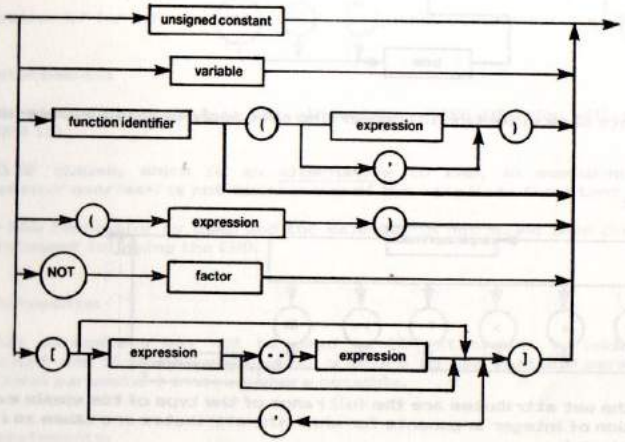
See Section 1.7.2 and Section 2 for more details of the use of dynamic variables and Appendix 4 for an example.

When specifying elements of multi-dimensional arrays the programmer is not forced to

use the same form of index specification in the reference as was used in the declaration.

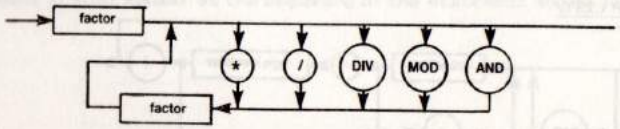
e.g. if variable a is declared as ARRAY[1..10] OF ARRAY[1..10] OF INTEGER then either a[1][1] or a[1,1] may be used to access element (1,1) of the array.

FACTOR.



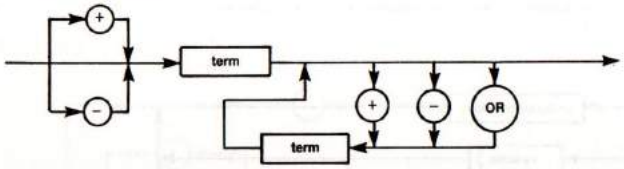
See EXPRESSION in Section 1.13 and FUNCTIONs in Section 3 for more details.

1.11 TERM.



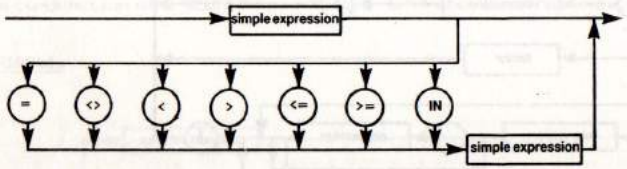
The lowerbound of a set is always zero and the set size is always the maximum of the base type of the set. Thus a SET OF CHAR always occupies 32 bytes (a possible 256 elements - one bit for each element). Similarly a SET OF 0..10 is equivalent to SET OF 0..255.

1.12 SIMPLE EXPRESSION.



The same comments made in Section 1.11 concerning sets apply to simple expressions.

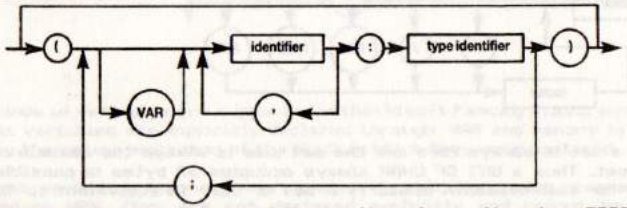
1.13 EXPRESSION.



When using IN, the set attributes are the full range of the type of the simple expression with the exception of integer arguments for which the attributes are taken as if [0..255] had been encountered.

The above syntax applies when comparing strings of the same length, pointers and all scalar types. Sets may be compared using >=, <=, <> or =. Pointers may only be compared using = and <>.

1.14 PARAMETER LIST.



A type identifier must be used following the colon - otherwise *ERROR* 44 will result.

Variable parameters as well as value parameters are fully supported.

Procedures and functions are not valid as parameters.

Files can only be designated variable parameters not value parameters.

1.15 STATEMENT.

Refer to the syntax diagram on page 14.

Assignment statements:

See Section 1.7 for information on which assignment statements are illegal.

CASE statements:

An entirely null case list is not allowed i.e. CASE OF END; will generate an error (*ERROR* 13).

The ELSE clause, which is an alternative to END, is executed if the selector ('expression' overleaf) is not found in one of the case lists ('constant' overleaf).

If the END terminator is used and the selector is not found then control is passed to the statement following the END.

FOR statements:

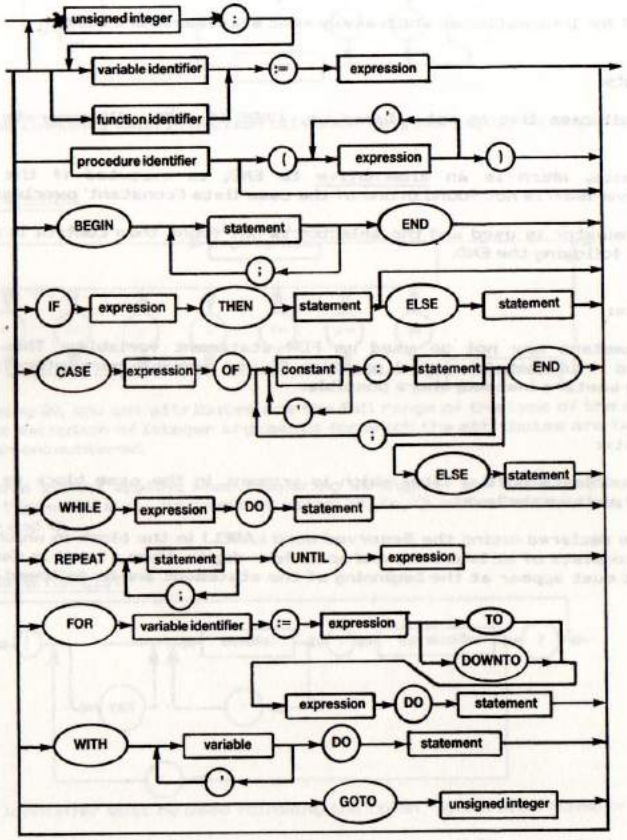
Variable parameters may not be used as FOR statement variables. This would be inefficient and it is certainly usual practice to use variable parameters sparingly, using local parameters instead, where possible.

GOTO statements:

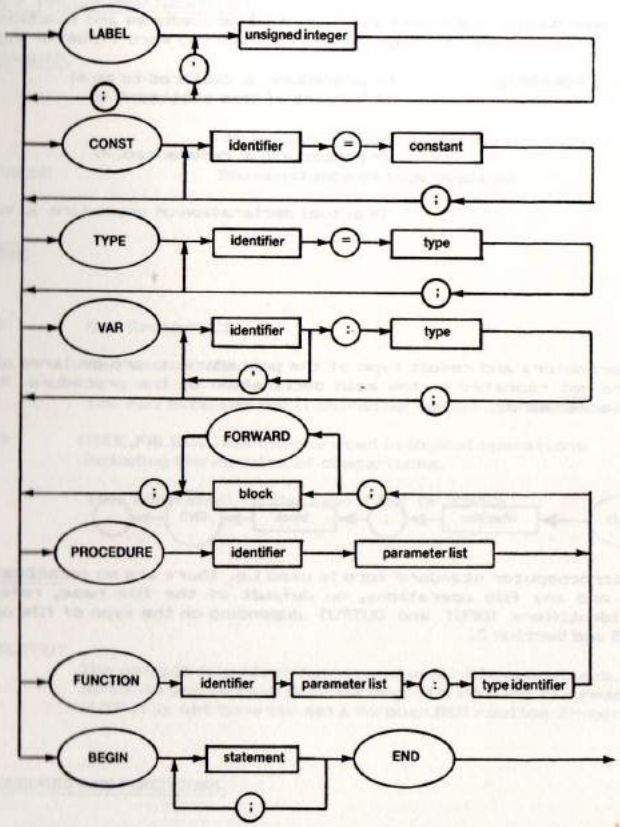
It is only possible to GOTO a label which is present in the same block as the GOTO statement and at the same level.

Labels must be declared (using the Reserved Word LABEL) in the block in which they are used; a label consists of at least one and up to four digits. When a label is used to mark a statement it must appear at the beginning of the statement and be followed by a colon - ':'.

STATEMENT.



1.16 BLOCK.



Note that, when a file variable is declared, then it may be followed, optionally, by a constant with a value between 1 and 255 inclusive enclosed in square brackets. This constant specifies the buffer size to be used for this file, in 128 character units. For example if you require the file file1 to have a buffer size of 2K (2048 characters) then the declaration should look like:

```
VAR     file1 : FILE OF CHAR [16];
```

or

```
CONST  filesize = 16;
VAR     file1 : TEXT[filesize];
```

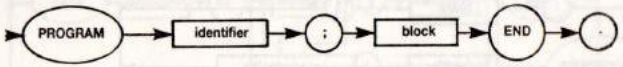
Forward References.

As in the Pascal User Manual and Report (Section 11.C.1) procedures and functions may be referenced before they declared through use of the Reserved Word FORWARD e.g.

```
PROCEDURE a(y:t) ; FORWARD;          (* procedure a declared to be *)
PROCEDURE b(x:t);                    (* forward of this statement *)
 BEGIN
 ....
 a(p);                               (* procedure a referenced. *)
 ....
 END;
PROCEDURE a;                         (* actual declaration of procedure a. *)
 BEGIN
 ....
 b(q);
 ....
 END;
```

Note that the parameters and result type of the procedure a are declared along with FORWARD and are not repeated in the main declaration of the procedure. Remember, FORWARD is a Reserved Word.

1.17 PROGRAM.



Note that the 'microcomputer standard' form is used i.e. there are no formal parameters of the program and any file operations, on default of the file name, refer to the predefined file identifiers INPUT and OUTPUT depending on the type of file operation. See Section 1.7.3 and Section 2.

SECTION 2 PREDEFINED IDENTIFIERS.

2.1 CONSTANTS.

MAXINT	The largest integer available i.e. 32767.
TRUE, FALSE	The constants of type Boolean.

2.2 TYPES.

INTEGER	See Section 1.3.
REAL	See Section 1.3.
CHAR	The full extended ASCII character set of 256 elements.
BOOLEAN	(TRUE,FALSE). This type is used in logical operations including the results of comparisons.
TEXT	This equivalent to the type FILE OF CHAR;.

2.3 VARIABLES.

INPUT, OUTPUT

The default file identifiers used in all file operations.
INPUT is set to read data through DOS routine 10 - read console buffer.
OUTPUT is set to write data through DOS routine 2 - normally the CRT.

2.4 PROCEDURES AND FUNCTIONS.

2.4.1 File Handling Procedures.

2.4.1.1 Preamble - the Buffer Variable f^.

Whenever a textfile f is declared then a variable f^, the buffer variable, is also created. f^, of type CHAR, can be thought of as a window through which we can access a component of the file - all data transfers to or from the file occur through f^. The buffer variable may be assigned or appear in an expression provided type compatibility is maintained.

The standard procedures GET and PUT are provided to handle the buffer variable on a primitive level while the procedures READ(LN) and WRITE(LN) allow more sophisticated input and output such as conversion of numbers to strings etc.

Before data is transferred between the file and the buffer variable, in either direction, the file must have been opened through the use of RESET (for reading) or REWRITE (for writing). These procedures provide the only method of opening a file.

The buffer size is, by default, 128 characters; this may be increased, in units of 128 characters, by enclosing a constant (giving the number of 128 character units) in square brackets after the declaration of the file f - see Section 1.16.

2.4.1.2 PUT(f)

PUT(f) appends the value of f^ to the file f provided that EOF is TRUE. After completion of the operation EOF remains TRUE and the buffer variable f^ becomes undefined. Note, therefore, that it is possible only to append a value to a textfile within Hisoft Pascal - files must be written sequentially.

Before PUT is used on a file, the file must have been opened for writing by the use of REWRITE - the exception to this is if the file is the default file OUTPUT which is con

sidered already open on entry to the program; a REWRITE must not be issued on the file OUTPUT.

2.4.1.3 REWRITE(f,fn)

REWRITE is used to open a textfile for writing. f specifies the file variable that is to be used within the program - this should have been declared in the outer block e.g. VAR file1 : FILE OF CHAR; remember that file variables may only be global. fn is ARRAY[1..14] OF CHAR and denotes the actual DOS file specification under which the file is to occur on the diskette. Note that fn must contain 14 characters i.e. 1 character for the drive name, then a colon, then 8 characters for the filename, then a period and finally 3 characters for the file extension; the drive name and colon may be replaced by 2 spaces in which case the default drive name is used. Examples:

```
REWRITE(file1,'a:TESTFILE.DAT');
```
creates ('makes') the file TESTFILE.DAT on drive A of the disk system.

```
REWRITE(file2,' LETTER .TXT');
```
creates the file LETTER.DAT on the default drive.

REWRITE causes any existing file fn to be deleted from the

diskette's directory - EOF(f) becomes TRUE and the buffer variable f^ is undefined.

2.4.1.4 WRITE

The procedure WRITE is used to append structured data to a textfile. The default textfile is the predefined file OUTPUT which is normally the standard output device of the computer. When the variable e to be written is simply of type character then WRITE(f,e) is exactly equivalent to: BEGIN f^ := e; PUT(f) END;.

Generally though, if f is a textfile:

WRITE((f,)P1,P2,.....Pn); is equivalent to:

```
BEGIN WRITE((f,)P1); WRITE((f,)P2); .....; WRITE((f,)Pn);
```

The write parameters P1,P2,.....Pn can have one of the following forms:

<e> or <e:m> or <e:m:n> or <e:m:H>

where e, m and n are expressions and H is a literal constant.

We have 5 cases to examine:

1) e is of type integer: and either <e> or <e:m> is used.

The value of the integer expression e is converted to a character string with a trailing space. The length of the string can be increased (with leading spaces) by the use of m which specifies the total number of characters to be output. If m is not sufficient for e to be written or m is not present then e is written out in full, with a trailing space, and m is ignored. Note that, if m is specified to be the length of e without the trailing space then no trailing space will be output.

2) e is of type integer and the form <e:m:H> is used.

In this case e is output in hexadecimal. If m=1 or m=2 then the value (e MOD 16^m) is output in a width of exactly m characters. If m=3 or m=4 then the full value of e is output in hexadecimal in a width of 4 characters. If m>4 then leading spaces are inserted before the full hexadecimal value of e as necessary. Leading zeroes will be inserted where applicable. Examples:

```
WRITE(1025:m:H);

m=1   outputs: 1
m=2   outputs: 01
m=3   outputs: 0401
m=4   outputs: 0401
```

m=5 outputs: _0401

3] e is of type real. The forms <e>, <e:m> or <e:m:n> may be used.

The value of e is converted to a character string representing a real number. The format of the representation is determined by n.

If n is not present then the number is output in scientific notation, with a mantissa and an exponent. If the number is negative then a minus sign is output prior to the mantissa, otherwise a space is output. The number is always output to at least one decimal place up to a maximum of 5 decimal places and the exponent is always signed (either with a plus or minus sign). This means that the minimum width of the scientific representation is 8 characters; if the field width m is less than 8 then the full width of 12 characters will always be output. If m>=8 then one or more decimal places will be output up to a maximum of 5 decimal places (m=12). For m>12 leading spaces are inserted before the number. Examples:

```
WRITE(-1.23E 10:m);

m=7   gives: -1.23000E+10
m=8   gives: -1.2E+10
m=9   gives: -1.23E+10
m=10  gives: -1.230E+10
m=11  gives: -1.2300E+10
m=12  gives: -1.23000E+10
m=13  gives: _-1.23000E+10
```

If the form <e:m:n> is used then a fixed-point representation of the number e will be written with n specifying the number of decimal places to be output. No leading spaces will be output unless the field width m is sufficiently large. If n is zero then e is output as an integer. If e is too large to be output in the specified field width then it is output in scientific format with a field width of m (see above). Examples:

```
WRITE(1E2:6:2)   gives:   100.00
WRITE(1E2:8:2)   gives:  _100.00
WRITE(23.455:6:1) gives:  _23.5
WRITE(23.455:4:2) gives:  _2.34550E+01
WRITE(23.455:4:0) gives:  _23
```

4] e is of type character or type string.

(If e is of type character then WRITE(f,e); is exactly equivalent to BEGIN f^ := e; PUT(f) END;.)

Either <e> or <e:m> may be used and the character or string of characters will be output in a minimum field width of 1 (for characters) or the length of the string (for string types). Leading spaces are inserted if m is sufficiently large.

5] e is of type Boolean.

Either <e> or <e:m> may be used and 'TRUE' or 'FALSE' will be output depending on the Boolean value of e , using a minimum field width of 4 or 5 respectively.

2.4.1.5 WRITELN

WRITELN((f)) appends an end of line marker to the textfile f.

WRITELN((f,)P1,P2,.....P3); is equivalent to:

BEGIN WRITE((f,)P1,P2,.....P3); WRITELN((f)) END;

The default file identifier f is the file OUTPUT. Remember that, for any WRITE or WRITELN on a textfile (other than OUTPUT) to be successful, then the file must first have been opened for writing through the use of REWRITE(f).

2.4.1.6 PAGE((f))

The procedure PAGE((f)) causes an ASCII form feed character (#0C) to be written to the file f. If f is absent then the file OUTPUT is assumed; this will cause the video screen to be cleared.

2.4.1.7 GET(f)

GET(f) advances the file window for textfile f by one position and then transfers one element of data from the textfile f to the associated buffer variable f^.

If EOF(f) is TRUE after the window has been advanced then the value of f^ after the GET(f) is CHR(26) and EOF remains TRUE.

Before GET(f) is used on a file the file must have been opened for reading through the use of RESET(f); the only exception is if the file is the default file INPUT which is considered already open on entry to the program - a RESET should not be issued on the file INPUT.

2.4.1.8 RESET(f,fn)

RESET(f,fn) is used to open a file for reading - the file identifier f, which should have been declared as a global variable, is associated with the DOS diskfile fn which must already exist. See Section 2.4.1.3 for details of the syntax of fn.

RESET(f,fn) opens the file f for reading and, if the file is not empty the first component of the textfile is assigned to the buffer variable f^ and EOF(f) becomes FALSE. If the file f is empty before the RESET(f) is issued then f^ is left undefined and EOF(f) becomes TRUE.

If the file is already open when RESET(f) is issued then the file is first closed before being opened for read operations.

RESET must not be used on the default file INPUT - this file is already open on entry to the program.

2.4.1.9 READ

The procedure READ is used to access data from textfiles. If the variable V to be read from the textfile f is simply of type character then READ(f,V); is exactly equivalent to: BEGIN V := f^; GET(f) END;. In general though:

READ(f,V1,V2,.....Vn); is equivalent to:

BEGIN READ(f,V1); READ(f,V2); .....; READ(f,Vn) END;

where V1, V2 etc. may be of type character, string, integer or real.

The default value of f is INPUT which is normally assigned to the system's keyboard.

The statement READ(f,V); has different effects depending on the type of V. There are 4 cases to consider:

1] V is of type character.

In this case READ(f,V); is exactly equivalent to: BEGIN V := f^; GET(f) END; and a character is effectively read from the buffer and assigned to V. If the text window on the file is positioned on a line marker (a CHR(13) character) then the function EOLN(f) will return the value TRUE and the buffer variable f^ will contain the value CHR(13). When a read operation is subsequently performed on the file the file window will be positioned at the start of a new line.

Important note: the default file INPUT initially contains a blank line so that EOLN is TRUE at the beginning of the program. This means that if the first read on the file INPUT is of type character then a CHR(13) value will be returned followed by the reading in of a new line from the keyboard; a subsequent read of type character will return the first character from this new line, assuming it is not blank. See also the procedure READLN below.

2] V is of type string.

A string of characters may be read using READ and in this case a series of characters will be read from the file until the number of characters defined by the string has been read or (EOLN) OR (EOF) = TRUE. If the string is not filled by the read (i.e. if end-of-line or end-of-file is reached before the whole string has been assigned) then the end of the string is filled with null (CHR(0)) characters - this enables the programmer to evaluate the length of the string that was read.

The note concerning the file INPUT in 1] above also applies here.

3] V is of type integer.

In this case a series of characters which represent an integer as defined in Section 1.3 is read. All preceding blanks and end-of-line markers are skipped (this means that integers may be read immediately from the default file INPUT cf. the note in 1] above).

If the integer read has an absolute value greater than MAXINT (32767) then the runtime error 'Number too large' will be issued and execution terminated.

If the first character read, after spaces and end-of-line characters have been skipped, is not a digit or a sign ('+' or '-') then the runtime error 'Number expected' will be reported and the program aborted.

4] V is of type real.

Here, a series of characters representing a real number according to the syntax of Section 1.3 will be read.

All leading spaces and end-of-line markers are skipped and, as for integers above, the first character afterwards must be a digit or a sign. If the number read is too large or too small (see Section 1.3) then an 'Overflow' error will be reported, if 'E' is present without a following sign or digit then 'Exponent expected' error will be generated and if a decimal point is present without a subsequent digit then a 'Number expected' error will be given.

Reals, like integers, may be read immediately from the default file INPUT; see 1] and 3] above.

2.4.1.10 READLN

READLN((f,)V1,V2,.....Vn); is equivalent to:

BEGIN READ((f,)V1,V2,.....Vn); READLN END;

READLN((f)) positions the file window over the component past the next end-of-line marker and assigns to f^ the value of this next component. Thus EOLN((f)) becomes FALSE after the execution of READLN((f)) unless the next line is blank.

The default value of f is INPUT.

READLN may be used to skip the blank line which is present at the beginning of the file INPUT i.e. for the file INPUT only, READLN has the effect of reading in a new buffer. This will be useful if you wish to read a component of type character from the beginning of INPUT - it is not necessary if you are reading an integer or a real from the first line in INPUT (since end-of-line markers are skipped) or if you are reading characters from subsequent lines.

2.4.2 File Handling Functions.

2.4.2.1 EOLN((f))

The function EOLN is a Boolean function which returns the value TRUE if the file window of file f is currently positioned over an end-of-line character (CHR(13)). Otherwise the function returns the value FALSE.

The default value of f is the file INPUT.

When EOLN((f)) is TRUE then the value of the associated buffer variable f^ is CHR(13).

2.4.2.2 EOF((f))

This is a Boolean function which returns the value TRUE when an end-of-file character (CHR(26)) appears under the file window of the file f. Otherwise EOF((f)) returns the value FALSE.

The default value of f is the file INPUT.

Note: EOF will become TRUE and remain TRUE when a CHR(26) character appears under the text window of any file but this will not prevent the action of any subsequent GET requests; this

enables any spurious end-of-file markers within a file to be skipped.

EOF can also be used to detect abnormal conditions during the writing of a file. During the writing of a file f then EOF(f) is normally TRUE. However if an abnormal condition, such as a full disk or directory, occurs then EOF(f) will become FALSE and the user can detect this.

2.4.2.3 INCH

The function INCH causes the standard input device (normally the keyboard) of the computer to be scanned and, if a key has been pressed, returns the character represented by the key pressed. If no key has been pressed then CHR(0) is returned. The function therefore returns a result of type character. You should use the compiler option C- with this function. (See Section 3).

2.4.3 Transfer Functions.

2.4.3.1 TRUNC(X)

The parameter X must be of type real or integer and the value returned by TRUNC is the greatest integer less than or equal to X if X is positive or the least integer greater than or equal to X if X is negative. Examples:

TRUNC(-1.5) returns -1 TRUNC(1.9) returns 1

2.4.3.2 ROUND(X)

X must be of type real or integer and the function returns the 'nearest' integer to X (according to standard rounding rules). Examples:

ROUND(-6.5) returns -6 ROUND(11.7) returns 12
ROUND(-6.51) returns -7 ROUND(23.5) returns 24

2.4.3.3 ENTIER(X)

X must be of type real or integer - ENTIER returns the greatest integer less than or equal to X, for all X. Examples:

ENTIER(-6.5) returns -7 ENTIER(11.7) returns 11

Note: ENTIER is not a Standard Pascal function but is the equivalent of BASIC's INT. It is useful when writing fast routines for many mathematical applications.

2.4.3.4 ORD(X)

X may be of any scalar type except real. The value returned is an integer representing the ordinal number of the value of X within the set defining the type of X.

If X is of type integer then ORD(X) = X ; this should normally be avoided.

Examples:

ORD('a') returns 97 ORD('@') returns 64

2.4.3.5 CHR(X)

X must be of type integer. CHR returns a character value corresponding to the ASCII value of X. Examples:

CHR(49) returns '1' CHR(91) returns '['

2.4.4 Arithmetic Functions.

In all the functions within this sub-section the parameter X must be of type real or integer.

2.4.4.1 ABS(X)

Returns the absolute value of X (e.g. ABS(-4.5) gives 4.5). The result is of the same type as X.

2.4.4.2 SQR(X)

Returns the value X*X i.e. the square of X. The result is of the same type as X.

2.4.4.3 SQRT(X)

Returns the square root of X - the returned value is always of

type real. A 'Maths Call Error' is generated if the argument X is negative.

2.4.4.4 FRAC(X)

Returns the fractional part of X: FRAC(X) = X - ENTIER(X).

As with ENTIER this function is useful for writing many fast mathematical routines. Examples:

FRAC(1.5) returns 0.5 FRAC(-12.56) returns 0.44

2.4.4.5 SIN(X)

Returns the sine of X where X is in radians. The result is always of type real.

2.4.4.6 COS(X)

Returns the cosine of X where X is in radians. The result is always of type real.

2.4.4.7 TAN(X)

Returns the tangent of X where X is in radians. The result is always of type real.

2.4.4.8 ARCTAN(X)

Returns the angle, in radians, whose tangent is equal to the number X. The result is of type real.

2.4.4.9 EXP(X)

Returns the value e^X where e = 2.71828. The result is always of type real.

2.4.4.10 LN(X)

Returns the natural logarithm (i.e. to the base e) of X. The result is of type real. If X <= 0 then a 'Maths Call Error' will be generated.

2.4.5 Further Predefined Procedures.

2.4.5.1 NEW(p)

The procedure NEW(p) allocates space for a dynamic variable. The variable p is a pointer variable and after NEW(p) has been executed p contains the address of the newly allocated dynamic variable. The type of the dynamic variable is the same as the type of the pointer variable p and this can be of any type except FILE.

To access the dynamic variable p^ is used – see Appendix 4 for an example of the use of pointers to reference dynamic variables.

To re-allocate space used for dynamic variables use the procedures MARK and RELEASE (see below).

2.4.5.2 MARK(v1)

This procedure saves the state of the dynamic variable heap to be saved in the pointer variable v1. The state of the heap may be restored to that when the procedure MARK was executed by using the procedure RELEASE (see below).

The type of variable to which v1 points is irrelevant, since v1 should only be used with MARK and RELEASE never NEW.

For an example program using MARK and RELEASE see Appendix 4.

2.4.5.3. RELEASE(v1)

This procedure frees space on the heap for use of dynamic variables. The state of the heap is restored to its state when MARK(v1) was executed – thus effectively destroying all dynamic variables created since the execution of the MARK procedure. As such it should be used with great care.

See above and Appendix 4 for more details.

2.4.5.4 INLINE(C1,C2,C3,.....)

This procedure allows Z80 machine code to be inserted within the Pascal program; the values (C1 MOD 256, C2 MOD 256, C3 MOD 256, ...) are inserted in the object program at the current location counter address held by the compiler. C1, C2, C3 etc. are integer constants of which there can be any number. Refer to Appendix 4 for an example of the use of INLINE.

2.4.5.5 USER(V)

USER is a procedure with one integer argument V. The procedure causes a call to be made to the memory address given by V. Since Hisoft Pascal holds integers in two's complement form (see Appendix 3) then in order to refer to addresses greater than #7FFF (32767) negative values of V must be used. For example #C000 is -16384 and so USER(-16384); would invoke a a call to the memory address #C000. However, when using a constant to refer to a memory address, it is more convenient to use hexadecimal.

The routine called should finish with a Z80 RET instruction (#C9) and must preserve the IX register.

2.4.5.6 HALT

This procedure causes program execution to stop with the message 'Halt at PC=XXXX' where XXXX is the hexadecimal memory address of the location where the HALT was issued. Together with a compilation listing, HALT may be used to determine which of two or more paths through a program are taken. This will normally be used during de-bugging.

2.4.5.7 POKE(X,V)

POKE stores the expression V in the computer's memory starting from the memory address X. X is of type integer and V can be of any type except FILE or SET. See Section 2.4.5.5 above for a discussion of the use of integers to represent memory addresses. Examples:

POKE(#6000,'A') places #41 at location #6000.
POKE(-16384,3.6E3) places 00 0B 80 70 at #C000.

2.4.5.8 OUT(i,c)

The procedure OUT has one parameter of type integer, i, and one of type char, c. The character c is output directly to the Z80 port number i. Example:

OUT(23,'A') outputs the value 'A' to port 23.

2.4.5.9. PRON

The parameterless procedure PRON causes output to be directed to the printer instead of the console. This simply uses DOS function 5 instead of 2.

2.4.5.10 PROFF

The parameterless procedure PROFF causes output to be directed to the console using DOS call 2.

2.4.5.11 PLOT(t,x,y)

The procedure PLOT has 3 parameters. The first is a boolean, t, which specifies whether the point given by the integer parameters, x,y, is to be lit or unlit. Thus PLOT(TRUE,X,Y) is equivalent to PLOT X,Y in BASIC and PLOT(FALSE,X,Y) is equivalent to UNPLOT(X,Y) in BASIC.

2.4.5.12 ORIGIN(x,y)

The procedure ORIGIN has two integer parameters and sets the graphics origin to (x,y). This corresponds directly to the BASIC command of the same name.

2.4.5.13 DRAW(x1,y1,x2,y2,z)

The procedure DRAW has 5 integer parameters. The first 4 specify the co-ordinates of the ends of the line to be drawn. The last parameter specifies the style of line as in BASIC. Thus DRAW(x1,y1,x2,y2,z) is equivalent to DRAW x1,y1 TO x2,y2,z in BASIC.

For example:

DRAW(0,0,60,60,2) draws a dotted line from (0,0) to (60,60).

2.4.5.14 FILL(x,y)

The procedure FILL has 2 integer parameters, x and y which give the co-ordinate to start filling the screen with the foreground graphics colour.

2.4.5.15 POLY(chords,startangle,finishangle,cx,cy,radx,rady,cinc,z)

The procedure POLY has one boolean paramater and 8 integer parameters and is used to draw polygons and ellipses.

The first parameter, chords, is of type boolean and is normally false. If it is true then lines are drawn from the centre of the polygon to the start and finish points.

The next two parameters startangle and finishangle must be between 0 and 1023. An angle of 0 corresponds to facing right and angles grow anti-clockwise with 1024 units in a whole circle. Thus 256 corresponds to upwards, 512 to leftwards and 768 to downwards.

If the startangle and finish angle are the same a complete polygon is produced. Note that due to the behaviour of the MOS routine if startangle and finishangle differ by one an almost full polygon is drawn.

The parameters cx and cy define the centre of the polygon. radx and rady define the horizontal and vertical radii of the polygon.

Note that because of the 4:3 aspect ratio of the screen rady/radx should be 4/3 to draw polygons that look regular.

The next parameter, cinc, determines how many sides the polygon has. To draw one with N sides use 1024 DIV N. To draw a circle or ellipse you should use a value of 32 or less. The lower the value the long plotting will take but a better circle will result.

The last parameter z determines the type of line that is drawn as in BASIC.

For example:

```
POLY( FALSE,128,128,100,100,60,80,256,0);
```

draws a square in the middle of the screen.

```
POLY( TRUE,0,256,80,80,80,120,10,2);
```

draws a quarter circle in the middle of the screen with chords and using dotted lines.

2.4.5.16 GCOL(foreground,background)

The GCOL procedure has two integer parameters. The first is the new graphics foreground colour and the second is the new graphics background colour. This corresponds directly to the BASIC

command of the same name. The corresponding colours are:

0 Transparent	8 Medium Red
1 Black	9 Light Red
2 Medium Green	10 Dark Yellow
3 Light Green	11 Light Yellow
4 Dark Blue	12 Dark Green
5 Light Yellow	13 Magenta
6 Dark Red	14 Grey
7 Cyan	15 White

e.g. GCOL(10,6) will cause any graphics following this command to be produced in Dark Yellow on a Dark Red background.

2.4.5.17 TCOL(foreground,background)

The TCOL procedure has two integer parameters. The first is the new text foreground colour and the second is the new text background colour. This corresponds directly to the BASIC TCOL command. See above for a list of colour numbers.

2.4.5.18 PSG(register,value)

The PSG procedure has two parammeters of type integer. The first is a sound register number (0..15) and the second is the value to output to the specified register in the Sound Generator.

2.4.6 Further Predefined Functions.

2.4.6.1 RANDOM

This returns a pseudo-random number between 0 and 255 inclusive. Although this routine is very fast it gives poor results when used repeatedly within loops that do not contain I/O operations.

If the user requires better results than this function yields then he/she should write a routine (either in Pascal or machine code) tailored to the particular application.

2.4.6.2 SUCC(X)

X may be of any scalar type except real and SUCC(X) returns the successor of X. Examples:

SUCC('A') returns 'B' SUCC('5') returns '6'

2.4.6.3 PRED(X)

X may be of any scalar type except real; the result of the function is the predecessor of X. Examples:

PRED('j') returns 'i' PRED(TRUE) returns FALSE

2.4.6.4 ODD(X)

X must be of type integer. ODD returns a Boolean result which is TRUE if X is odd and FALSE if X is even.

2.4.6.5 CPM(V1,V2)

This useful function has two integer parameters (V1 and V2) and returns an integer result. The effect is as follows: the Z80 register C is loaded with the value (V1 MOD 256) and the Z80 register pair DE is loaded with the value V2 and then a CALL 5 is executed; this is a subroutine call to the DOS with the routine number contained in register C and any extra information required by the routine contained in the register pair DE. Any result returned by the DOS routine will be returned as the result of the function CPM. For example:

DOS function 14 selects the disk drive whose number is contained in register E (0=drive A, 1=drive B etc.). Thus to select drive B from within a Hisoft Pascal program you can simply use: dummy := CPM(14,1).

DOS routine 25 returns the currently selected disk number so that to obtain this number in the Pascal variable disk you could use: disk := CPM(25,0);.

For further information on the DOS routines see the relevant DOS Interface Guide.

2.4.6.6 ADDR(V)

This function takes a variable identifier of any type as a parameter and returns an integer result which is the memory address of the variable identifier V. For information on how variables are held, at runtime, within Hisoft Pascal see Appendix 3. For an example of the use of ADDR see Appendix 4.

2.4.6.7 PEEK(X,T)

The first parameter of this function is of type integer and is used to specify a memory address (see Section 2.4.5.5). The second argument is a type; this is the result type of the function.

PEEK is used to retrieve data from the memory of the computer and the result may be of any type except FILE.

In all PEEK and POKE (the opposite of PEEK) operations data is moved in Hisoft Pascal's own internal representation detailed in Appendix 3. For example: if the memory from #5000 onwards contains the values: 50 61 73 63 61 6C (in hexadecimal) then:

```
WRITE(PEEK(#5000,ARRAY[1..6] OF CHAR)) gives 'Pascal'
WRITE(PEEK(#5000,CHAR)) gives 'P'
WRITE(PEEK(#5000,INTEGER)) gives 24912
WRITE(PEEK(#5000,REAL)) gives 2.46227E+29
```

see Appendix 3 for more details on the representation of types within Hisoft Pascal.

2.4.6.8 INP(p)

INP is a function with one integer parameter p. It returns as a character, the value that would be given by executing the Z80 instruction IN A,(p).

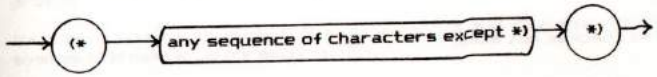
2.4.6.9 POINT(x,y)

This is a function which returns a boolean result and has two integer parameters. The result is true if the point given by the coordinate is lit and false if it is unlit.

SECTION 3 COMMENTS AND COMPILER OPTIONS.

3.1 Comments.

The following is the syntax for a comment:



A comment may occur between any two reserved words, numbers, identifiers or special symbols – see Appendix 2.

3.2 Compiler Options.

Compiler options are specified at the beginning of comments preceeded by a $. They consist of a letter followed by '+' or '-' except in the case of the 'F' option (see below). If more than one option is specified they should be separated by a comma.

Examples:

```
(*$L-   turn the listing off *)
(*$O-,C-,S-,A-,L- turn all checks off *)
```

The following options are available:

Option L:

Controls the listing of the program text and object code address by the compiler.

If L+ then a full listing is given.
If L- then lines are only listed when an error is detected.

DEFAULT: L+

Option O:

Controls whether certain overflow checks are made. Integer multiply and divide and all real arithmetic operations are always checked for overflow.

If O+ then checks are made on integer addition and subtraction routine calls are checked to ensure that their arguments are within range.

If O- then the above checks are not made.

DEFAULT: O+

Option C:

Controls whether or not keyboard checks are made during object code program execution. If C+ then if CTRLC is pressed then execution will return to DOS with a HALT message - see Section 2.4.5.6.

This check is made at the beginning of all loops, procedures and functions. Thus the user may use this facility to detect which loop etc. is not terminating correctly during the debugging process.

If C- then the above check is not made.

DEFAULT: C+

Option S:

Controls whether or not stack shecks are made.

If S+ then, at the beginning of each procedure and function call, a check is made to see if the stack will probably overflow in this block. If the runtime stack overflows the dynamic variable heap or the program then the message 'Out of RAM at PC=XXXX' is displayed and execution aborted. Naturally this is not foolproof; if a procedure has a large amount of stack usage within itself then the program may 'crash'. Alternatively, if a function contains very little stack usage while utilising recursion then it is possible for the function to be halted unnecessarily.

The address of the stack may be set at compilation time - see Section 1.2.

If S- then no stack checks are performed.

DEFAULT: S+

Option A:

Controls whether checks are made to ensure that array indices are within the bounds specified in the array's declaration.

If A+ and an array index is too high or too low then the message 'Index too high' or 'Index too low' will be displayed and the program execution halted.

If A- then no such checks are made.

DEFAULT: A+

Option I:

When using 16 bit 2's complement integer arithmetic, overflow occurs when performing a >, <, >=, or <= operation if the arguments differ by more than MAXINT (32767). If this occurs then the result of the comparison will be incorrect. This will not normally present any difficulties; however, should the user wish to compare such numbers, the use of I+ ensures that the results of the comparison will be correct. The equivalent situation may arise with real arithmetic in which case an overflow error will be issued if the arguments differ by more than approximately 3.4E38 ; this cannot be avoided.

If I- then no check for the result of the above comparisons is made.

DEFAULT: I-

Option P:

If P+ then the compiler listing is directed through the DOS call number 5; normally to a printer.

P- stops listing through DOS system call 5 and resumes normal listing through DOS system call number 2; normally to CRT.

DEFAULT: P-

Option F:

This option letter must be followed by a space and then a valid DOS file identifier of the form 'drive:filename.file extension' - if 'drive:' is omitted then the current default drive letter is assumed, the file extension must be ASC and this need not be specified.

The presence of this option causes inclusion of Pascal source text from the specified file from the end of the current statement - useful if the programmer wishes to build up a 'library' of much-used procedures and functions on the disk system and then include them in particular programs.

Example: {$F B:MATRIX include the text MATRIX.ASC from drive B};

This option may not be nested and may only appear within a program.

The compiler options may be used selectively. Thus debugged sections of code may be speeded up and compacted by turning the relevant checks off whilst retaining checks on untested pieces of code.

APPENDIX 1 ERRORS.

A.1.1 Error numbers generated by the compiler.

1. Number too large.
2. Semi-colon expected.
3. Undeclared identifier.
4. Identifier expected.
5. Use '=' not ':=' in a constant declaration.
6. '=' expected.
7. This identifier cannot begin a statement.
8. ':=' expected.
9. ')' expected.
10. Wrong type.
11. ':' expected.
12. Factor expected.
13. Constant expected.
14. This identifier is not a constant.
15. 'THEN' expected.
16. 'DO' expected.
17. 'TO' or 'DOWNTO' expected.
18. '(' expected.
19. Cannot write this type of expression.
20. 'OF' expected.
21. ',' expected.
22. ';' expected.
23. 'PROGRAM' expected.
24. Variable expected since parameter is a variable parameter.
25. 'BEGIN' expected.
26. Variable expected in call to READ.
27. Cannot compare expressions of this type.
28. Should be either type INTEGER or type REAL.
29. Cannot read this type of variable.
30. This identifier is not a type.
31. Exponent expected in real number.
32. Scalar expression (not numeric) expected.
33. Null strings not allowed (use CHR(0)).
34. '[' expected.
35. ']' expected.
36. Array index type must be scalar.
37. '..' expected.
38. ']' or ',' expected in ARRAY declaration.
39. Lowerbound greater than upperbound.
40. Set too large (more than 256 possible elements).
41. Function result must be type identifier.
42. ',' or ']' expected in set.
43. '..' or ',' or ']' expected in set.
44. Type of parameter must be a type identifier.
45. Null set cannot be the first factor in a non-assignment statement.
46. Scalar (including real) expected.
47. Scalar (not including real) expected.
48. Sets incompatible.
49. '<' and '>' cannot be used to compare sets.
50. 'FORWARD', 'LABEL', 'CONST', 'VAR', 'TYPE' or 'BEGIN' expected.

51. Hexadecimal digit expected.
52. Cannot POKE sets.
53. Array too large (> 64K).
54. 'END' or ';' expected in RECORD definition.
55. Field identifier expected.
56. Variable expected after 'WITH'.
57. Variable in WITH must be of RECORD type.
58. Field identifier has not had asociated WITH statement.
59. Unsigned integer expected after 'LABEL'.
60. Unsigned integer expected after 'GOTO'.
61. This label is at the wrong level.
62. Undeclared label.
63. Cannot assign or POKE files.
64. Can only use equality tests for pointers.
65. The parameter of this procedure/function should be of a FILE type.
66. File buffer too large (>= 256 records i.e. 32K).
67. The only write parameter for integers with two ':'s is e:m:H.
68. Strings may not contain end of line characters.
69. The parameter of NEW, MARK or RELEASE should be a variable of pointer type.
70. The parameter of ADDR should be a variable.
71. All files must be FILEs OF CHAR or subrange thereof.
72. Files may only be used as global variables or variable parameters.
73. RESET and REWRITE may not be used on INPUT or OUTPUT.

A.1.2 Runtime Error Messages.

When a runtime error is detected then one of the following messages will be displayed, followed by ' at PC=XXXX' where XXXX is the memory location at which the error occurred. Consult the compilation listing to see where in the program the error occurred, using XXXX to cross reference.

1. Halt
2. Overflow
3. Out of RAM
4. / by zero — also generated by DIV.
5. Index too low
6. Index too high
7. Maths Call Error
8. Number too large
9. Number expected
10. Line too long
11. Exponent expected
12. File Error *

Runtime errors result in the program execution being halted.

* File Error is given if there is an attempt is made to access a file which has not been opened for the required type of access. e.g. if a read on a file occurs after a REWRITE without an intervening RESET. The address given in 'PC=XXXX' is the address of the file variable rather than the location in the program where the error occurred. If it is not obvious which file has generated the error use the ADDR function. See Section 2.4.6.6.

APPENDIX 2 RESERVED WORDS AND PREDEFINED IDENTIFIERS.

A 2.1 Reserved Words.

AND	ARRAY	BEGIN	CASE	CONST	DIV	DO
DOWNTO	ELSE	END	FILE	FOR	FORWARD	FUNCTION
GOTO	IF	IN	LABEL	MOD	NIL	NOT
OF	OR	PACKED	PROCEDURE	PROGRAM	RECORD	REPEAT
SET	THEN	TO	TYPE	UNTIL	VAR	WHILE
WITH						

A 2.2 Special Symbols.

The following symbols are used by Hisoft Pascal and have a reserved meaning:

+	-	*	/			
=	<>	<	<=	>=	>	
(	)	[	]	(*	*)	^
:=	.	,	;	:	'	..

A 2.3 Predefined Identifiers.

The following entities may be thought of a declared in a block surrounding the whole program and they are therefore available throughout the program unless re-defined by the programmer within an inner block.
For further information see Section 2.

```
CONST       MAXINT = 32767;

TYPE        BOOLEAN = (FALSE, TRUE);
             CHAR {The expanded ASCII character set};
             INTEGER = -MAXINT..MAXINT;
             REAL {A subset of the real numbers. See Section 1.3.}
             TEXT = FILE OF CHAR;

VAR         INPUT,OUTPUT : TEXT;

PROCEDURE    WRITE; WRITELN; READ; READLN; RESET; REWRITE; GET; PUT;
             PAGE; HALT; USER; POKE; INLINE; NEW; MARK; RELEASE; OUT;
             PRON; PROFF; PLOT; DRAW; FILL; POLY; ORIGIN; GCOL; TCOL; PSG;

FUNCTION    ABS; SQR; ODD; RANDOM; ORD; SUCC; PRED; INCH; EOLN; EOF;
             PEEK; CHR; SQRT; ENTIER; ROUND; TRUNC; FRAC; SIN; COS;
             TAN; ARCTAN; EXP; LN; ADDR; CPM; INP; POINT;
```

APPENDIX 3 DATA REPRESENTATION AND STORAGE.

A 3.1 Data Representation.

The following discussion details how data is represented internally by Hisoft Pascal.

The information on the amount of storage required in each case should be of use to most programmers; other details may be needed by those attempting to merge Pascal and machine code programs.

A 3.1.1 Integers.

Integers occupy 2 bytes of storage each, in 2's complement form.
Examples:

```
   1   ≡   #0001
 256   ≡   #0100
-256   ≡   #FF00
```

The standard Z80 register used by the compiler to hold integers is HL.

A 3.1.2 Characters, Booleans and other Scalars.

These occupy 1 byte of storage each, in pure, unsigned binary.

Characters: 8 bit, extended ASCII is used.

```
'E' ≡ #45
'[' ≡ #5B
```

Booleans:

ORD(TRUE) = 1 so TRUE is represented by 1.
ORD(FALSE) = 0 so FALSE is representd by 0.

The standard Z80 register used by the compiler for the above is A.

A 3.1.3 Reals.

The (mantissa, exponent) form is used similar to that used in standard scientific notation - only using binary instead of denary. Examples:

$2 = 2*10^0$ or $1.0_2 * 2^1$

$1 = 1*10^0$ or $1.0_2 * 2^0$

$-12.5 \equiv -1.25*10^{1}$ or $-25*2^{-1}$

-11001_2*2^{-1}

-1.1001_2*2^{3} when normalised.

$0.1 \equiv 1.0*10^{-1}$ or $\frac{1}{10} \equiv \frac{1}{1010_2} \equiv \frac{0.1_2}{101_2}$

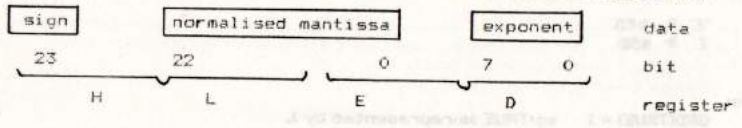
so now we need to do some binary long division..

$$
\begin{array}{r}
0.0001100 \\
101 \overline{)\,0.100000000000000} \\
\underline{101} \\
110 \\
\underline{101} \\
1000 \\
\underline{101} \\
\text{at this point we see that the fraction recurs}
\end{array}
$$

$$\frac{0.1_2}{101_2} = 0.000\dot{1}1\dot{0}0_2$$

$1.100\dot{1}1\dot{0}0_2 * 2^{-4}$ answer.

So how do we use the above results to represent these numbers in the computer? Well, firstly we reserve 4 bytes of storage for each real in the following format:



sign: the sign of the mantissa; 1 = negative, 0 = positive.
normalised mantissa: the mantissa normalised to the form 1.xxxxxx with the top bit (bit 22) always 1 except when representing zero (HL=0, DE=0).
exponent: the exponent in binary 2's complement form.

Thus:

2	≡	0 1000000	00000000	00000000	00000001	(#40, #00, #00, #01)
1	≡	0 1000000	00000000	00000000	00000000	(#40, #00, #00, #00)
-12.5	≡	1 1100100	00000000	00000000	00000011	(#E4, #00, #00, #03)
0.1	≡	0 1100110	01100110	01100110	11111100	(#66, #66, #66, #FC)

So, remembering that HL and DE are used to hold real numbers, then we would have to load the registers in the following way to represent each of the above numbers:

```
2     ≡   LD   HL,#4000
          LD   DE,#0100

1     ≡   LD   HL,#4000
          LD   DE,#0000

-12.5 ≡   LD   HL,#E400
          LD   DE,#0300

0.1   ≡   LD   HL,#6666
          LD   DE,#FC66
```

The last example shows why calculations involving binary fractions can be inaccurate; 0.1 cannot be accurately represented as a binary fraction, to a finite number of decimal places.

N.B. Reals are stored in memory in the order ED LH.

A 3.1.4 Records and Arrays.

Records use the same amount of storage as the total of their components.

Arrays: if n=number of elements in the array and
s=size of each element then

the number of bytes occupied by the array is n*s.

e.g. an ARRAY[1..10] OF INTEGER requires 10*2 = 20 bytes
an ARRAY[2..12,1..10] OF CHAR has 11*10=110 elements and so requires 110 bytes.

A 3.1.5 Sets.

Sets are stored as bit strings and so if the base type has n elements then the number of bytes used is: (n-1) DIV 8 + 1. Examples:

a SET OF CHAR requires (256-1) DIV 8 + 1 = 32 bytes.
a SET OF (blue, green, yellow) requires (3-1) DIV 8 + 1 = 1 byte.

A 3.1.6 Files.

Files require (41 + 128*buffer size) bytes where buffer size is specified in square brackets ([]) after the file variable declaration (default is 1) - see Section 1.16.
Each file has associated with it a File Information Block (FIB); this is of the form given overleaf:

0-1	Pointer to the data buffer; f^ in Pascal terminology.
2-3	End of buffer address (set up on creation).
4	Status 0=unused (set up on creation). 1=read. -1=write.
5	EOF(f)
6	Drive number: 0=default 1=A,2=B,3=C etc., -1=terminal (i.e. INPUT and OUTPUT).

The following is the CP/M FCB entry - see the DOS manual for details.

7	Zero.
8-15	Filename.
16-18	File type.
19-22	Zeroes.
23-38	CP/M disk map.
39	Record Count i.e. initialised to 0.
40-	Data Buffer.

For the file INPUT we have:

0-6	as above.
7	maximum linelength - 80 characters.
8	number of characters in the current line.
9-88	Data Buffer.

For the file OUTPUT we have

0-6	as above.
7	one byte buffer.

A 3.1.7 Pointers.

Pointers occupy 2 bytes which contain the address (in Intel format i.e. low byte first) of the variable to which they point.

A 3.2 Variable Storage at Runtime.

There are 3 cases where the user needs information on how variables are stored at runtime:

a. Global variables	- declared in the main program block.
b. Local variables	- declared in an inner block.
c. Parameters and returned values.	- passed to and from procedures and functions.

These individual cases are discussed below and an example of how to use this information may be found in Appendix 4.

Global variables

Global variables are allocated from the top of the runtime stack downwards e.g. if the runtime stack is at #B000 and the main program variables are:

```
VAR     i : INTEGER;
        ch : CHAR;
        x : REAL;
```

then:

i (which occupies 2 bytes - see the previous section) will be stored at locations #B000-2 and #B000-1 i.e. at #AFFE and #AFFF.

ch (1 byte) will be stored at location #AFFE-1 i.e. at #AFFD.

x (4 bytes) will be placed at #AFF9, #AFFA, #AFFB and #AFFC.

Local variables

Local variables cannot be accessed via the stack very easily so, instead, the IX register is set up at the beginning of each inner block so that (IX-4) points to the start of the block's local variables e.g.

```
PROCEDURE test;
VAR i,j : INTEGER;
```

then:

i (integer - so 2 bytes) will be placed at IX-4-2 and IX-4-1 i.e. IX-6 and IX-5.
j will be placed at IX-8 and IX-7.

Parameters and returned values

Value parameters are treated like local variables and, like these variables, the earlier a parameter is declared the higher address it has in memory. However, unlike variables, the lowest (not the highest) address is fixed and this is fixed at (IX+2) e.g.

```
PROCEDURE test(i : REAL; j : INTEGER);
```

then:

j (allocated first) is at IX+2 and IX+3.
i is at IX+4, IX+5, IX+6, and IX+7.

Variable parameters are treated just like value parameters except that they are

always allocated 2 bytes and these 2 bytes contain the address of the variable e.g.

```
PROCEDURE test(i : INTEGER; VAR x : REAL);
```

then:

the reference to x is placed at IX+2 and IX+3; these locations contain the address where x is stored. The value of i is at IX+4 and IX+5.

Returned values of functions are placed above the first parameter in memory e.g.

```
FUNCTION test(i : INTEGER) : REAL;
```

then i is at IX+2 and IX+3 and space is reserved for the returned value at IX+4, IX+5, IX+6 and IX+7.

APPENDIX 4 SOME EXAMPLE HISOFT PASCAL 4 PROGRAMS.

There follow some example programs written in Hisoft Pascal. All have been thoroughly tested and therefore can be typed in with confidence.

Some of the programs may be of practical use; certainly all of them demonstrate particular aspects of the implementation of Pascal within Hisoft Pascal, to facilitate the user's learning process.

```
(* A program to create a file 'TESTDATA.DAT' on disk drive A.
Shows the handling of strings versus individual characters
and use of FILE, REWRITE, EOLN, READLN *)

PROGRAM MAKEFILE;

CONST
   MAXFILE = 10;                              (* file entries *)
   MAXSTRING = 12;                            (* max. number length *)

VAR
   DATA : FILE OF CHAR;
   I : 1..MAXFILE;
   i : 1..MAXSTRING;
   CH : CHAR;
   CHSTRING : ARRAY[1..MAXSTRING] OF CHAR;

BEGIN
  REWRITE(DATA,'A:TESTDATA.DAT');             (* open file for writing *)
  FOR I := 1 TO MAXFILE DO
    BEGIN
      WRITE('Name please? ');
      READLN;
      WHILE NOT EOLN DO                       (* process input character *)
        BEGIN                                 (* by character *)
          READ(CH);
          WRITE(DATA,CH)                      (* and output to file *)
        END;                                  (* character by character *)
      WRITELN(DATA);                          (* new line to file *)

      WRITE('Number please? ');
      READLN;                                 (* process input line as *)
      READ(CHSTRING);                         (* as whole string *)
      i:=1;
      WHILE CHSTRING[i] <> CHR(0) DO
       BEGIN
        WRITE(DATA,CHSTRING[i]);              (* write string to file, *)
        i:=i+1                                (* character by character. *)
       END;
      WRITELN(DATA)
    END
END.
```

```
(* Program to list the lines of a file in reverse order.
   Example of: Pointers (to create a linked-list), Files and strings. *)

PROGRAM FILEREVERSE;                          (* Reverses entries in a file. *)

CONST
   STRLEN=20;                                 (* Maximum string length *)

TYPE
   STRING = PACKED ARRAY[1..STRLEN] OF CHAR;  (* Note: PACKED ignored *)
   ID = RECORD
          NEXT : ^ID;                         (* Although illegal in  *)
          NAME,NUMBER : STRING                (* Standard Pascal, the recursive
                                                 reference to ID *)
        END;                                  (* allows a linked list *)
   LINK = ^ID;                                (* structure to be created. *)

VAR
  PREVIOUS, CURRENT : LINK;                   (* Pointers to ID. *)
  DATA : TEXT;

BEGIN
  RESET(DATA,'  TESTDATA.DAT');              (* Open 'TESTDATA.DAT' on the
                                               default drive, for reading. *)
  PREVIOUS := NIL;
   WHILE NOT EOF(DATA) DO
    BEGIN
      NEW(CURRENT);                           (* Creates a new dynamic variable
                                                 of type ID. *)
      WITH CURRENT^ DO                        (* Get ready to assign to the new
                                                 dynamic variable. *)

        BEGIN
         READLN(DATA,NAME);                   (* Read in the name and number *)
         READLN(DATA,NUMBER);                 (* from separate lines. *)
         NEXT := PREVIOUS                     (* Make link point to previous *)
        END;                                  (* entry. *)

      PREVIOUS := CURRENT                     (* Make this entry the entry
                                                 for the next iteration. *)
    END;

(* Having initialised the linked list, now output it in reverse order. *)

  CURRENT := PREVIOUS;
  WHILE CURRENT<>NIL DO
    BEGIN
      WITH CURRENT^ DO
        WRITELN(NAME,'    ',NUMBER);
      CURRENT := CURRENT^.NEXT
    END
END.
```



```
(* Program to list lines of a file in reverse order.
 Shows use of pointers, records, EOF, MARK and RELEASE. *)

PROGRAM ReverseLine;

TYPE elem=RECORD                              (* create linked-list structure *)
           next:^elem;
           ch:CHAR
         END;
     link=^elem;

VAR prev,cur,heap:link;                       (* all pointers to 'elem' *)
    data : FILE OF CHAR;

BEGIN
  RESET(data,'A:TESTDATA.DAT');               (* open file for reading. *)
  REPEAT                                      (* until end-of-file *)
    MARK(heap);                               (* assign top of heap to 'heap'. *)
    prev:=NIL;                                (* points to no variable yet. *)
    WHILE NOT EOLN(data) DO
     BEGIN
      NEW(cur);                               (* create a new dynamic record *)
      READ(data,cur^.ch);                     (* and assign its field to one
                                                 character from file. *)
      cur^.next:=prev;                        (* this field's pointer addresses *)
      prev:=cur                               (* previous record. *)
     END;

(* Write out the file entry backwards by scanning the
 records set up backwards. *)

    cur:=prev;
    WHILE cur <> NIL DO                       (* NIL is first. *)
     BEGIN
      WRITE(cur^.ch);                         (* WRITE this field i.e. character. *)
      cur:=cur^.next                          (* Address previous field. *)
     END;
    WRITELN;
    RELEASE(heap);                            (* Release dynamic variable space. *)
    READLN(data)
  UNTIL EOF(data)
END.
```

```
(* Program to show how to 'get your hands dirty'!
   i.e. how to modify Pascal variables using machine code.
   Demonstrates PEEK, POKE, ADDR and INLINE. *)

PROGRAM divmult2;

VAR r:REAL;

FUNCTION divby2(x:REAL):REAL;          (* Function to divide by 2 ..
                                          .. quickly. *)

VAR i:INTEGER;
BEGIN
  i:=ADDR(x)+1;                        (* Point to the exponent of x. *)
  POKE(i,PRED(PEEK(i,CHAR)));          (* Decrement the exponent of x.
                                          see Appendix 3.1.3. *)

  divby2:=x
END;

FUNCTION multby2(x:REAL):REAL;         (* Function to multiply by 2 ..
                                          .. quickly. *)

BEGIN
 INLINE(#DD,#34,3);                    (* INC (IX+3) - the exponent of x
                                          - see Appendix 3.2. *)

 multby2:=x
END;

BEGIN
 REPEAT
  WRITE('Enter the number r ');
  READ(r);                             (* No need for READLN - see
                                          Section 2.4.1.9. *)

  WRITELN('r divided by two is',divby2(r):7:2);
  WRITELN('r multiplied by two is',multby2(r):7:2)
 UNTIL r=0
END.
```



```
(* Program to show use of recursion. *)

 PROGRAM FACTOR;

(* This program calculates the factorial of a number input from the
 keyboard  1) using recursion  and  2) using an iterative method. *)

TYPE
  POSINT = 0..MAXINT;

VAR
  METHOD : CHAR;
  NUMBER : 0..MAXINT;

(* Recursive algorithm. *)

FUNCTION RFAC(N : POSINT) : INTEGER;

  VAR F : POSINT;

  BEGIN
    IF N>1 THEN F := N * RFAC(N-1)             (* RFAC invoked N times. *)
           ELSE F:=1;
    RFAC := F
  END;

(* Iterative solution. *)

FUNCTION IFAC(N : POSINT) : INTEGER;

  VAR I,F : POSINT;

  BEGIN
    F := 1;
    FOR I := 2 TO N DO F := F*I;               (* Simple loop. *)
    IFAC := F
  END;

BEGIN
  REPEAT
    WRITE('Give method (I OR R) and number  ');
    READLN;
    READ(METHOD,NUMBER);
    IF METHOD = 'R'
       THEN WRITELN(NUMBER,'! = ',RFAC(NUMBER))
       ELSE WRITELN(NUMBER,'! = ',IFAC(NUMBER))
  UNTIL NUMBER=0
END.
```

BIBLIOGRAPHY.

K. Jensen and N. Wirth — PASCAL USER MANUAL AND REPORT. Springer-Verlag 1975.

I.R. Wilson and A.M. Addyman — A PRACTICAL INTRODUCTION TO PASCAL WITH BS 6192. MacMillan 1982.

W. Findlay and D. A. Watt — PASCAL. AN INTRODUCTION TO SYSTEMATIC PROGRAMMING. Pitman Publishing 1978.

J. Tiberghien — THE PASCAL HANDBOOK. SYBEX 1981.

J. Welsh and J. Elder. — INTRODUCTION TO PASCAL.

HISOFT
180 High Street North
Dunstable, Beds. LU6 1AT
Tel: (0582) 696421

We hope that you enjoy using our Pascal compiler and welcome any correspondence from you, whether it be constructive criticism or bug-reporting. We especially need to know of any problems that you may experience so that we can continually improve the compiler.

To complement Hisoft Pascal on the Tatung Einstein computer we also have available DEVPAC, our powerful assembler and disassembler/debugger. The assembler contains many extremely advanced features not found in other less powerful products.

a) Macros with parameters are supported. DEVPAC allows you to set up library files of macros on disc which it will search at assembly-time.

b) DEVPAC can include source text from disc while assembling. This important and powerful feature enables very large source files to be assembled and removes the annoying memory constraints that plague other assemblers. (160K of source no problem).

c) DEVPAC uses a true binary-tree symbol table which greatly increases the speed of assembly (to more than 4000 lines per minute).

d) Length of labels is under user control enabling use of meaningful symbols and optimisation of symbol table size.

e) Complex arithmetic operations possible for label definitions.

f) The monitor/dissassembler while including all the normal functions (single-step, dynamic breakpoints, read in disc-files etc) includes <continue until breakpoint x times> for loop repetitions, enter ASCII characters, find sequence of bytes OR find assembler instruction (eg.. find LD HL, C000) or part instruction (eg..find PUSH or find LD BC,).

g) ALL registers and contents shown and easily changed. Bytes following register address, contents of memory and 20 line dissassembly displayed.

We think that if you compare the features of DEVPAC with any other Z80 development packages available for the Einstein you will find that when it comes to assembly language... there is no choice.

We also have all our products available for a wide range of home micros; Spectrum, Sharp, Memotech, MSX, Amstrad, CP/M etc. Write for details.

Please feel free to use the space overleaf for your notes....

NOTES